

# Python for Data Science

## 3. Functions



# Contents

---

- 3.1**     **Defining Functions**
- 3.2**     **Functions with Multiple Parameters**
- 3.3**     **Python Standard Library**
- 3.4**     **Functional-Style Programming**

# 1 Defining Functions

## function

A **function definition** begins with the `def` keyword, followed by the *function name*, a set of *parentheses* and a *colon* (:).

The required parentheses contain the function's **parameter list** – a comma-separated list of parameters representing the data that the function needs to perform its task.

The indented lines after the colon (:) are the function's **block**, which consists of an optional docstring followed by the statements that perform the function's task.

```
In [1]: def square(number):  
...:     """Calculate the square of number."""  
...:     return number ** 2  
...:
```

```
In [2]: square(7)  
Out[2]: 49
```

```
In [3]: square(2.5)  
Out[3]: 6.25
```

## return statement

When a function finishes executing, it returns control to its caller. In *square*'s block, the `return` statement:

```
return number ** 2
```

first squares *number*, then terminates the function and gives the result back to the caller.

There are two other ways to *return control* from a function to its caller:

- Executing a `return` statement without an expression terminates the function and *implicitly* returns the value `None` to the caller.
- When there's no `return` statement in a function, it *implicitly* returns the value `None` after executing the last statement in the function's block.

# 1 Defining Functions

## Calling a Function

The expression `square(7)` passes the *argument* 7 to `square`'s *parameter* `number`.

Then `square` calculates `number ** 2` and *returns* the result.

The parameter `number` exists only during the function call. It's created on each **call** to the function to receive the *argument* value, and it's destroyed when the function returns its result to the caller.

A function's *parameters* and *variables* defined in its block are all **local variables** – they can be used only inside the function and exist only while the function is executing.

Trying to access a local variable outside its function's block causes a `NameError`, indicating that the variable is not defined.

## 2

## Functions with Multiple Parameters

```
In [1]: def maximum(value1, value2, value3):  
....:     """Return the maximum of three values."""  
....:     max_value = value1  
....:     if value2 > max_value:  
....:         max_value = value2  
....:     if value3 > max_value:  
....:         max_value = value3  
....:     return max_value  
....:
```

```
In [2]: maximum(12, 27, 36)
```

```
Out[2]: 36
```

```
In [3]: maximum(12.3, 45.6, 9.7)
```

```
Out[3]: 45.6
```

```
In [4]: maximum('yellow', 'red', 'orange')
```

```
Out[4]: 'yellow'
```

You also may call `maximum` with mixed types, such as `int` and `float`

## 2

# Functions with Multiple Parameters

**min**  
**max**

Built-in `max` and `min` functions know how to determine the largest and smallest of their two or more arguments:

```
In [6]: max('yellow', 'red', 'orange', 'blue', 'green')
Out[6]: 'yellow'
```

```
In [7]: min(15, 9, 27, 14)
Out[7]: 9
```

**Python**  
**Standard**  
**Library**

Using built-in functions or functions from the *Python Standard Library's* **modules** rather than writing your own can reduce development time and increase program reliability, portability and performance.

For a list of Python's built-in functions and modules, see <https://docs.python.org/3/library/>

## 2

# Functions with Multiple Parameters

### Default Parameter

When defining a function, you can specify that a parameter has a **default parameter value**.

When calling the function, if you *omit* the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed.

```
In [1]: def rectangle_area(length=2, width=3):  
...:     """Return a rectangle's area."""  
...:     return length * width  
...:
```

```
In [2]: rectangle_area()  
Out[2]: 6
```

```
In [3]: rectangle_area(10)  
Out[3]: 30
```

```
In [4]: rectangle_area(10, 5)  
Out[4]: 50
```



## 2

# Functions with Multiple Parameters

### Keyword Arguments

When calling functions, you can use **keyword arguments** to pass arguments in *any* order.

Each keyword *argument in a call* has the form *parametername=value*.

```
In [1]: def rectangle_area(length, width):  
...:     """Return a rectangle's area."""  
...:     return length * width  
...:
```

```
In [2]: rectangle_area(width=5, length=10)  
Out[3]: 50
```

## 2

# Functions with Multiple Parameters

### Arbitrary Argument Lists

Functions with **arbitrary argument lists**, such as built-in functions `min` and `max`, can receive *any* number of arguments.

```
In [1]: def average(*args):  
...:     return sum(args) / len(args)  
...:
```

```
In [2]: average(5, 10)  
Out[2]: 7.5
```

```
In [3]: average(5, 10, 15)  
Out[3]: 10.0
```

```
In [4]: average(5, 10, 15, 20)  
Out[4]: 12.5
```

```
In [5]: grades = [88, 75, 96, 55, 83]
```

```
In [6]: average(*grades)  
Out[6]: 79.4
```

## 2

# Functions with Multiple Parameters

## Local Scope

Each identifier has a **scope** that determines where you can use it in your program. For that portion of the program, the identifier is said to be “in scope.”

A local variable’s identifier has **local scope**.

It’s “in scope” only from its definition to the end of the function’s block.

It “goes out of scope” when the function returns to its caller.

## Global Scope

Identifiers defined outside any function (or class) have **global** scope – these may include functions, variables and classes.

Variables with global scope are known as **global variables**.

Identifiers with global scope can be used in a `.py` file or interactive session anywhere after they’re defined.

# 3

## Functional-Style Programming

### Functional-Style

Python offers “functional-style” features that help you write code which is less likely to contain errors, more concise and easier to read, debug and modify.

Functional-style programming lets you simply say *what* you want to do. It hides many details of *how* to perform each task.

### Pure Functions

A pure function’s result depends only on the argument(s) you pass to it. When you call the pure function, it does not modify its argument.

```
In [1]: values = [1, 2, 3]
```

```
In [2]: sum(values)
```

```
Out[2]: 6
```

```
In [3]: sum(values)  # same call always returns same result
```

```
Out[3]: 6
```

```
In [4]: values
```

```
Out[5]: [1, 2, 3]
```